Misc 1.1.5
Internals Manual

Department RIM — Internal Report

Jean-Jacques Girardot
girardot@emse.fr
March 2006

École Nationale Supérieure des Mines de Saint-Etienne
158 Cours Fauriel
42023 Saint-Étienne Cédex

Working draft. Release 0.27

Printed 30 mars 2009
Misc internal documentation. This document contains 45 pages.

Documentation is like sex; when it's good, it's very, very good,

and when it's bad, it's better than nothing.

– Dick Brandon

Chapter 1

Introduction

1.1 What is MISC ?

1.1.1 A few words

MISC is a very small implementation of the SCHEME language (hence the name : MISC stands for "Micro scheme") written in Java. Its primary goal was to explain to students what is SCHEME and how to implement it. Although MISC provides only a subset of the SCHEME language, it can be used as an extension language in applications written in Java. Generally, the MISC primitives tend to conform to the norm of Scheme as described in [3].

This manual [1] describes the internals of the Misc system. An other manual [2] describes the language and its main functions.

1.1.2 About this manual

This is the first version of the manual describing the internals of MISC. It is far from being complete. Readers are invited to send their comments to the author, at `girardot@emse.fr`.

MISC was developed on a PC laptop, operating under Linux. This document itself was written using LyX [5], a document editor which provides a quasi-WYSIWYG mode for LaTeX [4].

1.2 A short introduction to MISC

1.2.1 Getting the system

MISC is available from its Home page, at :

`http://kiwi.emse.fr/Misc/`

MISC distribution is a *compressed tar* file, which includes all the java source code, some examples, a test suite, and some documentation.

1.2.2 Installation

Misc is distributed as a compressed tar file (currently `SchV1.1.5.tar.gz`), which contains a main directory with some documentation and various related files, including **Misc.java** and **MiscApplet.java**,

two classes showing examples of use of MISC, and a subdirectory, **Sch**, which contains the java classes implementing the system. Java must of course be installed on your machine.

To install MISC, do :

```
gunzip -c SchV1.1.5.tar.gz | tar xf -
```

and then, inside the `SchV1.1.5` directory, type :

```
make
```

The command :

```
java Misc
```

runs the interpreter, if the `CLASSPATH` variable contains either the `SchV1.1.5` directory, or a dot.

Misc can also be run as an applet. The command is :

```
appletviewer RunMiscApplet.html
```

The file `RunMiscApplet.html` contains a reference to the `MiscApplet` class.

Chapter 2

Description of the system

2.1 General concepts

This document more or less makes the assumption that the reader is familiar with the Scheme language and its implementation. Although Scheme implementations are roughly compatible, and their concepts certainely are, the implementations are quite different. This first section therefore describes the general aspects of the implementation.

2.1.1 Finality and design

Misc has been written to act as an extension (or script, or embedded) language for applications written in Java. While Java is in itself a very dynalic and flexible language, there are cases where some additional flexibility is needed, like dynamically interacting with an application to modify its behaviour in ways that where not intended by the conceptor[1].

As such, Misc implements the minimum set of functions required to program in Scheme, and some additional ways to interact with the embedding application.

2.1.2 Data types

Misc uses a subset of the Scheme data types : boolean, symbols, characters, strings, integers, null and pairs. It also provides support for any embedded Java objects.

Pairs are the essential support of the intepreter. They are used to represent most data types, and also the internal representation of Scheme programs, which is a «slightly» compiled version of the source text. The representation of pairs is discussed in 2.3 on page 7, and the interpretation process in 2.4 on page 16.

2.2 Structure of the interpreter

2.2.1 The class Sch

The central class of the Misc interpreter is **Sch**. This class handles :

- the initialization of the whole system ;

[1] I know, a virus also does that...

- the management of the REPL (Read Eval Print Loop).
- the handling of program exceptions.

Moreover, it provides the following class methods :

public static String schinit (int) this provides a general way to initialize the system. The parameter is an integer, the *mode*, which is 0 if MISC is to be used as a stand-alone application, or 1 if it is to be used as an Applet. This method also returns a message (a string) giving the name and version of the interpreter.

public static void load (String) this method (valid only when `mode==1`) loads a scheme expression or sequence of expressions, provided in their external representation as a character string, into the interpreter.

public static String result () this method (valid only when `mode==1`) returns the printed characters accumulated so far during the execution of the interpreter as a character string. It also clears the internal buffer used to accumulate the results.

public static void loop () this method performs the actual evaluation of the expressions loaded in the scheme interpreter (when `mode==1`), or read from the input device (when `mode==0`). Control returns to the caller only when there is nothing left to execute.

public static void end () this method «closes» the MISC interpreter and prints an end message. It may, in future version, release some of the structures of the interpreter.

2.2.2 The class Globaldata

The class **Globaldata** is used to define the «global» constants and variables used in the interpreter. The following constants are bytes or integers, and use the following naming convention :

BC... the evaluator opcodes, which are also specific scheme cells ;

C... some specific cell numbers of the interpreter, used to represent special objects such as the *null*, *true* and *false*, the *undefined value*, etc.

ERR... the numbers associated with error messages ;

RC... the evaluator return codes.

RF... the run flags.

T... the data-types of the interpreter.

TR... the trace flags.

Some other «constants» act as parameters that can be «modified» if necessary. This includes the following :

KSSTACK Initial size of control and value stacks

KSMSTACK Maximum size of control and value stacks

KSGDM Initial number of memory cells

KSMGDM Maximum size of memory, in number of cells

KSEVTCC Default thread tick-count

2.2.3 The class Misc

The class **Misc** provides an example of a stand alone application using **Sch**. What it actually does is just call successively *schinit*, *loop* and *end*.

2.2.4 The class MiscApplet

The class **MiscApplet** provides an example of an applet using **Sch**. The file **RunMiscApplet.html** can be installed on an HTTP server to run this applet, or tested independently of a server with the applet viewer, by :

```
appletviewer RunMiscApplet.html
```

2.3 Memory Management

2.3.1 Introduction

Efficiency in a Lisp or Scheme system is in a large part related to the capacity to allocate cells efficiently. This in turn depends of the representation of the cells themselves.

2.3.1.1 About the representation of cells

Two main approaches can be used to represent cells :

1. Use Java objects. This is a solution which is simple, oriented in the spirit of Java, but rather inefficient :
   - creating a new cell implies the creation of a new Java object. This implies the cost of allocating memory to represent the cell and of initializing the cell data structure ;
   - freeing unused cells is automatic (it happens when the number of references to a cell becomes equal to zero). However, some scheme data structures (like "A points to B, and B points to A") may be never released.
2. Represent cells as part of a unique large Java data structure (typically an array). This is the approach used in Misc.
   - allocating a new cell is very efficient. Basically, it is :

     ```
     newCell = freeList;
     freeList = Gdm.cellcdr[freeList];
     ```

   - freeing unused cells must be explicitly programmed ; the work is done by a *garbage collector*. Timings show that this represents only a small overhead in the application.

2.3.1.2 Structures of the Memory Manager

The Misc «memory» is handled by the **Gdm** class. Cells are represented by the union of the four arrays :

```
byte [] celltyp;
int [] cellcar;
int [] cellcdr;
Object [] cellobj;
```

An individual cell is represented by its index in these arrays. The cell «n» is therefore the conceptual association of the four values `celltyp[n]`, `cellcar[n]`, `cellcdr[n]` and `cellobj[n]`. Since the arrays are static fields of the **Gdm** class, these fields are referenced in any external program as `Gdm.celltyp[n]`, `Gdm.cellcar[n]`, `Gdm.cellcdr[n]` and `Gdm.cellobj[n]`. We will sometimes use the terms «type of a cell», «car of a cell», «cdr of a cell» and «object of a cell» to reference these four fields.

2.3.2 The data types of MISC

All data types of MISC are represented by cells. While this is clearly expensive in the case of «simple» data types such as integers, that are very often allocated and destroyed, it is a practical approach that greatly simplifies all the run-time procedures.

In the following description, all the mentioned constants are defined in the class **Globaldata**.

2.3.2.1 Null

There is a unique *null* value, representing the empty list. The number of the corresponding cell is CNULL (actually, the value of this constant is 0, because this optimizes slightly the Java code generated, but any other value could do). The *null* value uses the following representation :

celltyp[CNULL] TNULL (a constant) ;

cellcar[CNULL] (unused) ;

cellcdr[CNULL] (unused) ;

cellobj[CNULL] (unused).

Note that no cell other than CNULL should have the type TNULL, and no program should ever generate such a cell. When it is necessary to return an empty list in a java method, the program *must* return CNULL.

2.3.2.2 Undefined value

There is a unique cell representing the *undefined value*. The number of this cell is CUNDEF. Loading such a value in the evaluator produces an *undefined value error*.

celltyp[CUNDEF] TUNDEF (a constant) ;

cellcar[CUNDEF] (unused) ;

cellcdr[CUNDEF] (unused) ;

cellobj[CUNDEF] (unused).

Just as in the case of the empty list, a method should not generate an object of type TUNDEF, but should return CUNDEF.

2.3.2.3 Booleans

The two values #t (*true*) and #f (*false*) are represented by the two specific cells numbered CTRUE and CFALSE. Their representations are :

celltyp[CTRUE] TBOOL (a constant) ;

cellcar[CTRUE] (unused) ;

cellcdr[CTRUE] (unused) ;

cellobj[CTRUE] (unused).

and :

celltyp[CFALSE] TBOOL (a constant) ;

cellcar[CFALSE] (unused) ;

cellcdr[CFALSE] (unused) ;

cellobj[CFALSE] (unused).

As can be seen, the representations of the values *true* and *false* are identical. Testing if a value is *true* or *false* is achieved by testing if the corresponding cell number is equal to CTRUE or CFALSE. Just as in the case of the empty list, a method should not generate an object of type TBOOL, but should return either CTRUE or CFALSE. Actually, no method ever tests the equality to CTRUE, but only to CFALSE, since any *non false* scheme value is considered as *true*.

2.3.2.4 Integers

Integers use the following representation :

celltyp[n] TINT (a constant) ;

cellcar[n] the value of the 32 bits java integer ;

cellcdr[n] (unused) ;

cellobj[n] (unused).

Most operations on integers are handled by the class **MathsOps**. A specific method to create integers is provided in the class **Gdm** :

public static int newint(int) the parameter is a 32 bits java integer (the value to be represented), and the result is a Misc cell representing the integer. The content of the fields of such a cell should not be changed. Note that different cells may represent the same integer. From a language point of view, these cells are **`eqv?`** but not **`eq?`**.

A method should not modify the value of an integer. If a method must return an integer value, it must create a new integer, but never change the `cellcar[n]` value. However, a method can return any already existing integer value.

2.3.2.5 Characters

Characters use the following representation :

celltyp[n] TCHAR (a constant) ;

cellcar[n] the 16 bits Unicode representation of the character, extended to a 32 bits integer ;

cellcdr[n] (unused) ;

cellobj[n] (unused).

A specific method to create characters is provided in the class **Gdm** :

public static int newchar(int) the parameter is a 32 bits java integer (the character to be represented, extended to a 32 bits integer), and the result is a Misc cell representing the character. The content of the fields of such a cell should not be changed. Note that different cells may represent the same character. From a language point of view, these cells are **`eqv?`** but not **`eq?`**.

A method should not modify the value of a character. If a method must return a character value, it must create a new character, but never change the `cellcar[n]` value. However, a method can return any already existing character value.

2.3.2.6 Strings

Strings use the following representation :

celltyp[n] TSTR (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] (unused) ;

cellobj[n] a *String* java object, representing the Misc string.

Most operations on strings are handled by the class **StringsOps**. A specific method to create strings is provided in the class **Gdm** :

public static int newstring(String) the parameter is a java String, and the result is a Misc cell representing the string. The content of the fields of such a cell should not be changed. Note that different cells may represent the same string. From a language point of view, these cells are **`eqv?`** but not **`eq?`**.

2.3.2.7 Subroutines

This is the designation for system primitives. These items use the following representation :

celltyp[n] TSUBR (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] an integer, the designation of the primitive inside its class.

cellobj[n] a java object, a member of the java class implementing the primitive.

System subroutines are defined in different classes that inherit of the class **SchPrimitive** (see § 3.1.2, p. 30). New subroutines can be created at run time by loading classes defining subroutines, with the primitive `load-misc-module`. See § 3.3, p. 34 for a description of this facility.

2.3.2.8 Symbols

Symbols use the following representation :

celltyp[n] TSYM (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] (unused) ;

cellobj[n] a *String* java object, representing the print name of the symbol.

The management of symbols is handled by the class **Symbols**. A program should never generate a cell of type TSYM, but should call the appropriate method of class **Symbols** :

public static int get(String) the parameter is a java String representing the print name of the symbol ; the returned value is the cell number representing the symbol.

2.3.2.9 Pairs

Pairs use the following representation :

celltyp[n] TPAIR (a constant) ;

cellcar[n] a reference to any cell ;

cellcdr[n] a reference to any cell ;

cellobj[n] (unused).

Pairs are managed by the class **Gdm**. Almost any class in the system manipulates pairs. Most of the primitive functions operating on pairs are defined in classes **CROps** and **ConsOps**, but many other classes, such as **FnsOps**, **LangOps**, etc, create pairs. The following methods are available to create pairs :

public static int newcons () this method creates a new cell of type *pair*. The *car* and *cdr* fields are initialized to CNULL.

public static int newcons (byte) this method creates a new cell of the type specified by the parameter. The *car* and *cdr* fields are initialized to CNULL.

public static int newcons (byte, int, int) this method creates a new cell of the type specified by the first parameter. The *car* and *cdr* fields are initialized with the values of the second and third parameters respectively.

public static int cons (int, int) this method creates a new pair, with the *car* and *cdr* fields initialized with the values of the first and second parameters respectively.

The fields `cellcar[n]` and `cellcdr[n]` of any cell can be modified by any java mathod.

2.3.2.10 Vectors

Vectors are sequences of Scheme items. They use the following representation :

celltyp[n] TVECT (a constant) ;

cellcar[n] null ;

cellcdr[n] null ;

cellobj[n] a reference to a java object of type **SchVector**. Such an object contains a positive integer, `vsize`, which represents the size of the vector, and an array of integers, `vals`, which represents the elements themselves. The elements are *pairs* (c.f. 2.3.2.9 on the preceding page), and are represented by their index in the arrays `celltyp`, `cellcar`, `cellcdr`, and `cellobj`.

2.3.2.11 Environments

Environments use the following representation :

celltyp[n] TENV (a constant) ;

cellcar[n] *null* (§ 2.3.2.1), or a list of bindings, i.e. a proper list whose elements are value cells (§ 2.3.2.14) ;

cellcdr[n] *null*, or a reference to the parent environment ;

cellobj[n] (unused).

2.3.2.12 Lambda expressions

Lambda expressions use the following representation :

celltyp[n] TLAMBDA (a constant) ;

cellcar[n] a reference to a pair, whose *car* is the formal arguments of the lambda expression (either a symbol or a list of symbols), and whose *cdr* is the lexical environment of the function) ;

cellcdr[n] the code of the function, usually a list ;

cellobj[n] (unused).

2.3.2.13 Comments

Comments constitute a data type which can be read by the primitive procedure **`read-token`** when the reading of comments is enabled (otherwise, comments are treated as white spaces). Comments use the following representation :

celltyp[n] TCOMM (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] (unused) ;

cellobj[n] a *String* java object, representing the content of the comment, including the starting semicolon.

2.3.2.14 Value cells

Value cells are used to represent *bindings*, i.e. symbol-value pairs. They use the following representation :

celltyp[n] TVC (a constant) ;

cellcar[n] a reference to a symbol ;

cellcdr[n] a reference to a scheme value ;

cellobj[n] (unused) ;

The reason why value cells are not represented as pairs is to prevent accidental modifications of bindings.

2.3.2.15 Control Codes, or Opcodes

Control codes are the «*byte codes*» of the **Evaluator** (see § 2.4.3, p. 24). They use the following representation :

celltyp[n] TCTRL (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] (unused) ;

cellobj[n] (unused).

Control codes are distinguished by the **Evaluator** on their unique cell numbers, defined as constants in the class **Globaldata**.

2.3.2.16 Objects

Objects are cells that can designate a Java object. The reference to the java object is kept in the field `cellobj[n]`. Objects use the following representation :

celltyp[n] TOBJ (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] (unused) ;

cellobj[n] a reference to a java object.

A specific method to create objects is provided in the class **Gdm** :

public static int newcons (Object) this method creates a new cell of type object. The *car* and *cdr* fields are initialized to CNULL, and the *cellobj* field receives a reference to the java Object passed as a parameter.

Note that different Misc data types can reference an object in their *cellobj* field. This is the case for *strings*, *symbols*, *ports*, and for some other specialized data types, such as *methods*, *threads*, etc.

2.3.2.17 Free cells

Free cells are «used» only by the memory management. They are arranged as a list, pointed by the variable `cellfree` in **Gdm**. Free cells use the following representation :

celltyp[n] TFREE (a constant) ;

cellcar[n] (unused) ;

cellcdr[n] the number of the next free cell, or -1 if this cell is the last one of the free cells list ;

cellobj[n] (unused).

Free cells should never appear in a data structure or during the evaluation process. In other terms, no java class other than **Gdm** is allowed to create and manipulate free cells.

2.3.3 Garbage collection

The *garbage collection* is the process by which unused cells are reclaimed and made again available to the MISC programs. The garbage collection is triggered either by an explicit call to the method *gc* of the **Gdm** class, either by an implicit call, when a method such as *newcons* is called to allocate a new memory cell, and no free cell is available. The garbage collection can be explicitly invoked inside a scheme program by the procedure :

(gc) This operation activates the garbage collector, which collects unused cells, and returns the number of cells available. New cells are dynamically added when the number of free cells is less than a percentage of the total number of cells after a garbage collection.

```
? (gc)
=>  7127
```

2.3.3.1 Algorithms

The garbage collector uses a mark and sweep algorithm, which operates in two phases.

The mark phase During the first phase, all cells that are supposed to be permanently referenced by the interpreter are marked. Marking consists in «or-ing» the byte representing the type of the cell with the value 128.

The method **gc()** of the class Gdm marks all the cells of the system whose number is less than a constant (CLAST), then explores cells whose number is comprised between CLAST and another constant (CFREE), and a little set of variables (`p0`, `p1` to `p7`, `q0`, `q1` to `q7`). «Exploring» means that the cells themselves are marked, and that the cells which they reference are also explored.

The sweep phase During this second phase, the system sets the value of the variable `cellfree` to -1, then sweeps all the cells, starting from the last one down to zero. If the cell is marked, its mark bit is reset. If the cell is not marked, this means that the cell has not been reached during the mark phase, and therefore is no more used by the system. The contents of the cell is reset to a standard value (a *free cell*, see § 2.3.2.17, p. 14), some cleaning is done on the object field of the cell, and the cell is added to the free list.

Adding the cell to the free list consists of :

- setting the *cdr* of the cell to the value of `cellfree` ;
- setting the value of `cellfree` to the number of the cell being freed.

2.3.3.2 Programming in spite of the garbage collector

Since the garbage collection is a process than can be started at any time during the execution of a Misc primitive (actually, at any moment a new cell is created), an algorithm written in Java may be in the position where some cells have already been created, but these cells have not been stored in Misc structure. The cells are therefore in the risk of being reclaimed by the garbage collector. In such a case, it is necessary to «protect» these cells.

Protecting a cell can be achieved by putting the number of the cell in a place that is explored during the *mark* phase. This is the case of variables `p0` to `p7`, `q0` to `q7`. These variables should not be used directly, but only through this specific procedure :

public static void protect1 (int) this method of the class **Gdm** protects its parameter for a «short period of time» (the idea is that the last eight cells passed as parameters to *protect1* are actually protected). This means that when less than 5 or 6 cells are needed for an algorithm, this method can be used to protect these cells.

When the number of cells to protect is unknown, the procedure *protect* can be used to that purpose :

public static void protect (int) this method of the class **Gdm** introduces its parameter (the number of a cell) into a list of cells to be protected. The whole list is freed the next time the system enters the REPL loop.

2.3.3.3 Testing new algorithms

When an algorithm creates new cells, each creation of a new cell may trigger a garbage collection. The garbage collection may reclaim cells that have just been allocated by the algorithm, but not yet inserted in protected structure. It is a good idea to test new operations in a mode of operation that systematically triggers a garbage collection each time a new cell is requested. This can be obtained, in a Scheme program, by calling the procedure *debug* with the parameter 1 :

(debug 0) reset to «standard» mode.

(debug 1) enter a mode when the garbage collection is triggered at each new allocation of a cell ;

(debug 2) enter a mode when the garbage collector is triggered before each execution of a primitive ;

The procedure modifies the value of the variable *debug* defined in the class **Globaldata**.

2.4 The Evaluation process

The evaluator is perhaps the most important process of the MISC system. It is implemented by the class **Evaluator**. The MISC evaluator operates on an internal representation of scheme programs generated by the class **Comp**.

2.4.1 The representation of MISC programs

MISC uses an internal representation for Scheme programs based on lists. In such lists, all data-types can appear, with the following interpretation :

Control codes are the opcodes for the Misc virtual machine (see § 2.3.2.15, page 13 for the general description of control codes, and § 2.4.3, page 24 for an exhaustive list). Each opcode corresponds to a specific action taken by the evaluator. Most of the opcodes correspond to actions needed to implement scheme special forms.

Symbols represent unresolved bindings of primitive and defined variables (see § 2.3.2.8, page 11).

Value cells represent resolved bindings (see § 2.3.2.14, page 13).

Lists represent code segments.

Other data-types are considered as «constants». This includes integers, characters, strings, but also functions, ports, etc.

This representation is generated by the class **Comp**.

2.4.1.1 Compilation of scheme programs

The compilation process (which is defined here as a transformation from a classical lisp data structure to an interpretable form) is performed by the *comp* method, defined as :

```
public static int comp(int exp, int lvl, int env)
```

The three parameters represent the code to compile, a list of local variables related to this code, and the compile environment. Here is an example of the compiled form for a Misc expression (the compiled form is represented as printed by the interpreter when run with *trace* equal to 4) :

```
? (abs (* 3 (- 2 (max 3 7 6)) 4 5))
-> ([F1] {abs} ([F4] {*} 3 ([F2] {-} 2 ([F3] {max} 3 7 6)) 4 5))
=> 300
```

The compiler uses the following algorithm :

a symbol is looked for in the local variables list. If it appears in the list, it is considered as a binding that is to be dynamically resolved at execution time, and the symbol itself is provided as the result of the compilation. This is the case of the parameters of a lambda expression. If the symbol does not appear in the list, it is looked for in the environment. If it is not found in the environment, a new binding is created in the user global environment. In any case, the result is a pointer to a binding, i.e. a value cell.

a constant is left unchanged. Constants that can appear in programs are nulls, integers, characters, booleans and strings.

a list is analyzed ; if its first item, in the current environment, resolves to a keyword of the language, a compilation process specific to this keyword takes place. Otherwise, each element of the list is compiled, and the result is arranged as a new list suitable as an input to the **Evaluator**. Specifically, if E_i denotes a scheme expression and C_i its compiled form, the list $(E_0\ E_1\ E_2 \ldots E_n)$ is compiled as $(F_n\ C_0\ C_1\ C_2 \ldots C_n)$, where F_n is the specific opcode denoting an n-parameter function call (for n less than 9 ; names of these constants are **BCF00**, **BCF01**, up to **BCF08** ; for n greater than 8, the generated code is $(N\ Fn\ C_0\ C_1\ C_2 \ldots C_n)$, where N is an integer representing the value n, and Fn is the constant **BCFnn**. Examples of codes :

```
? (max 2 4 5)
-> ([F3] {max} 2 4 5)
=> 5
? (+ 1 2 3 4 5 6 7 8 9 10 11 12)
-> (12 [Fn] {+} 1 2 3 4 5 6 7 8 9 10 11 12)
=> 78
```

The notation `[F0]` represents the constant **BCF00**, and so on.

2.4.1.2 Generated code

This part presents the generated code for the different scheme constructs. The notation `[CODE]` represents a specific *opcode* of the MISC evaluator, and C_i is the compiled code for E_i. The compiled form is represented as printed by the interpreter when run with *trace* equal to 4.

- **and** construct

Scheme form	Internal representation
`(and)`	`#t`
`(and` E_1`)`	C_1
`(and` E_1 E_2`)`	`(`C_2 `[AND]` C_1`)`
`(and` E_1 E_2 `...` E_n`)`	`(((...(`C_n `[AND] ...` C_3`) [AND]` C_2`) [AND]` C_1`)`

The notation `[AND]` represents the constant **BCAND**.

```
? (and 1 2 3)
-> ((3 [AND] 2) [AND] 1)
=> 3
```

- **begin** construct

Scheme form	Internal representation
`(begin)`	`()`
`(begin` E_1`)`	C_1
`(begin` E_1 E_2`)`	`(`C_2 `[POP]` C_1`)`
`(begin` E_1 E_2 `...` E_n`)`	`(`C_n `[POP] ...` C_3 `[POP]` C_2 `[POP]` C_1`)`

The notation `[POP]` represents the constant **BCPOPV**.

```
? (begin 1 2 3 4)
-> (4 [POP] 3 [POP] 2 [POP] 1)
=> 4
```

- **if** construct

Scheme form	Internal representation
`(if` E_1 E_2`)`	`(`C_2 `[AND]` C_1`)`
`(if` E_1 E_2 E_3`)`	`(`C_3 C_2 `[IF]` C_1`)`

The notation `[IF]` represents the constant **BCIF**. The notation `[AND]` represents the constant **BCAND**.

```
? (if 1 2)
-> (2 [AND] 1)
=> 2
? (if 1 2 3)
-> (3 2 [IF] 1)
=> 2
```

- **or** construct

Scheme form	Internal representation
`(or)`	`#f`
`(or` E_1`)`	C_1
`(or` E_1 E_2`)`	`(`C_2 `[OR]` C_1`)`
`(or` E_1 E_2 `...` E_n`)`	`(((...(`C_n `[OR]` `...` C_3`) [OR]` C_2`) [OR]` C_1`)`

The notation `[OR]` represents the constant **BCOR**.

```
? (or 1 2 3)
-> ((3 [OR] 2) [OR] 1)
=> 1
```

- **quote** construct

Scheme form	Internal representation
`(quote data)`	`(data [QUOTE])`

The notation `[QUOTE]` represents the constant **BCQUOTE**.

```
? (quote (a b c))
-> ((a b c) [QUOTE])
=> (a b c)
```

- **cond** construct

Scheme form	Internal representation
`(cond (else` E_1`))`	C_1
`(cond (`E_1 E_2`))`	`(`C_2 `[AND]` C_1`)`
`(cond (`E_1 `=>` E_2`))`	`(#f` C_2 `[=>]` C_1`)`
`(cond (`E_1 E_2`)` E_c`)`	`(`C_c C_2 `[IF]` C_1`)`
`(cond (`E_1 `=>` E_2`)` E_c`)`	`(`C_c C_2 `[=>]` C_1`)`

The notation E_c represents the "rest" of a cond construct, and C_c is its compiled form. The notation `[AND]` represents the constant **BCAND**. The notation `[=>]` represents the constant **BCCOND**. The notation `[IF]` represents the constant **BCIF**.

```
? (cond (1 2) (3 4) (5 6))
-> (((6 [AND] 5) 4 [IF] 3) 2 [IF] 1)
=> 2
```

- **case** construct

Scheme form	Internal representation
(case E (L_1 E_1))	([FALSE] C_1 L_1 [CASE] C)
(case E (else E_1))	(C_1 [POP] C)
(case E (L_1 E_1) (L_2 E_2))	(([FALSE] C_2 L_2 [CASE]) C_1 L_1 [CASE] C)
(case E (L_1 E_1) (else E_2))	((C_2 [POP]) C_1 L_1 [CASE] C)
(case E (L_1 E_1) E_c)	(C_c C_1 L_1 [CASE] C)

The notation [POP] represents the constant **BCPOPV**. The notation [CASE] represents the constant **BCCASE**. The notation [FALSE] represents the constant **BCFALSE**. The notation E_c represents the "rest" of a case construct, and C_c is its compiled form. It consists of embedded forms similar to the ones that appears in lines 3 and 4 of the table : depending on the last clause (*test* or else), its form is one of :

(((...([FALSE] C_n L_n [CASE])...) C_3 L_3 [CASE]) C_2 L_2 [CASE])
(((...(C_n [POP])...) C_3 L_3 [CASE]) C_2 L_2 [CASE])

An example :

```
? (case 'd ((a b) 1) ((c d) 2) (else 3))
-> (((3 [POP]) 2 (c d) [CASE]) 1 (a b) [CASE] (d
[QUOTE]))
=> 2
```

- **define** construct

Scheme form	Internal representation
(define var E_1)	({var} [DEFINE] C_1)
(define var)	({var} [DEFINE] ())

The notation [DEFINE] represents the constant **BCDEF**.

```
? (define toto 4)
-> ({toto} [DEFINE] 4)
=> toto
```

- **set!** construct

Scheme form	Internal representation
(set! var E_1)	({var} [SET] C_1)

The notation [SET] represents the constant **BCSET**.

```
? (set! toto 7)
-> ({toto} [SET] 7)
=> 7
```

- **lambda** construct

Scheme form	Internal representation
(lambda V E)	(V (begin E) [LAMBDA])

The notation [LAMBDA] represents the constant **BCLAMBDA**. The notation V represents a single identifier, or a list of identifiers. The notation E represents a single expression or a list of expressions. Note that no «compilation» of the inner expression is done at that time ; compilation occurs when the [LAMBDA] opcode is executed.

```
? (lambda (x) (+ x 3))
-> ((x) (begin (+ x 3)) [LAMBDA])
=> #<LAMBDA ((x) . #<ENV:1270:76>) ([F2] {+} x 3)>
```

- **delay** construct

Scheme form	Internal representation
(delay E)	([DELAY] (() (begin E) [LAMBDA]))

The notation [LAMBDA] represents the constant **BCLAMBDA**. The notation [DELAY] represents the constant **BCDELAY**. Note that no «compilation» of the inner expression is done at that time ; compilation occurs when the [DELAY] opcode is executed.

```
? (delay (+ 2 3))
-> ([DELAY] (() (begin (+ 2 3)) [LAMBDA]))
;=>  #<delay:1255>
```

- **let** construct

The following tables show some examples of the let construct. The first row corresponds to a Scheme expression, the second to the internal representation of this expression.

A let construct with no local variable :

(let () E)
((C) [LET])

This let construct has a single local variable :

(let ((L_1 E_1)) E)
((C L_1 [ADDENV] C_1) [LET])

This let construct has two local variables :

(let ((*L*1 *E*1) (*L*2 *E*2)) *E*)
((*C* *L*1 [ADDENV] *L*2 [ADDENV] *C*2 *C*1) [LET])

Finally, this is the general form of a let construct :

(let ((*L*1 *E*1) (*L*2 *E*2) ... (*Ln* *En*)) *E*)
((*C* *L*1 [ADDENV] *L*2 [ADDENV] ... *Ln* [ADDENV] *Cn* ... *C*2 *C*1) [LET])

The notation `[LET]` represents the constant **BCLET**. The notation `[ADDENV]` represents the constant **BCADENV**. Note that the internal representation allows to compute the values of local variables before these variables are added to the environment.

```
? (let ((a 5) (b (+ 3 4))) (+ 2 a b))
-> ((([F3] {+} 2 a b) a [ADDENV] b [ADDENV]
           ([F2] {+} 3 4) 5) [LET])
;=>  14
```

- **letrec** construct

  The following tables show some examples of the letrec construct. The first row corresponds to a Scheme expression, the second to the internal representation of this expression.

  A letrec construct with no local variable :

(letrec () *E*)
((*C*) () [LETREC])

This let construct has a single local variable :

(letrec ((*L*1 *E*1)) *E*)
((*C* *L*1 [SETU] *C*1) (*L*1) [LETREC])

This let construct has two local variables :

(letrec ((*L*1 *E*1) (*L*2 *E*2)) *E*)
((*C* *L*1 [SETU] *C*1 *L*2 [SETU] *C*2) (*L*2 *L*1) [LETREC])

Finally, this is the general form of a let construct :

(letrec ((*L*1 *E*1) (*L*2 *E*2) ... (*Ln* *En*)) *E*)
((*C* *L*1 [SETU] *C*1 *L*2 [SETU] *C*2 ... *Ln* [SETU] *Cn*) (*Ln* ... *L*2 *L*1) [LETREC])

The notation `[LETREC]` represents the constant **BCLETREC**. The notation `[SETU]` represents the constant **BCSETU**. The list, to the left of the `[LETREC]` constant, is the local variable list, which is introduced in the environment before computing the values of these local variables.

```
? (letrec ((a 5) (b 8)) (+ 2 a))
-> ((([F2] {+} 2 a) a [SETU] 5 b [SETU] 8) (b a) [LETREC])
;=>  7
```

2.4.2 The stack handling procedures

The evaluator works with two stacks : the *control stack*, and the *value stack*. These stacks contain references to cells, which are non negative integers. The control stack is used to hold the input expression, and more generally, all code expressions that will be used as an input by the evaluator. The value stack holds intermediate and final results. The size of the stacks is initially limited (to 16 entries, the value of the constant *KSSTACK*), but can grow as necessary up to a fixed limit (the *KSMSTACK* constant). Four operations are provided to handle these stacks :

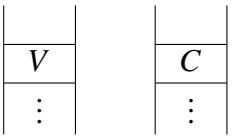
int pop () this operation returns the value which is on the top of the value stack, and decrements the stack pointer. The method throws the *ERRSTKE* exception (*stack empty*), if the stack is empty.

int popCtl () this operation returns the value which is on the top of the control stack, and decrements the stack pointer. The method throws the *ERRSTKE* exception (*stack empty*), if the stack is empty.

void push (int cell) this operation pushes a new value onto the value stack. The method throws the *ERRSTKF* exception (*stack full*) if increasing the size of the stack would overflow the maximum fixed size (*KSMSTACK*).

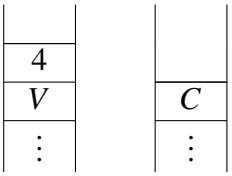
void pushCtl (int cell) this operation pushes a new value onto the value stack. The method throws the *ERRSTKF* exception (*stack full*) if increasing the size of the stack would overflow the maximum fixed size (*KSMSTACK*).

We use in this document the following convention to represent the stacks :



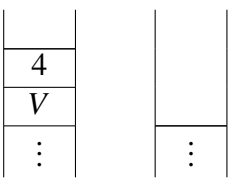
Value Control

Item V is the top of the value stack, and item C the top of the control stack. $\vdots$ represents the other items of the stack. After the execution of `push(4)`, the stacks become :



Value Control

After the execution of `popCtl()`, which returns the value C, the configuration becomes :



Value Control

2.4.3 The Evaluator

The evaluator is implemented by the class **Evaluator**. The evaluator works with the *control* and the *value stacks*. Generally speaking, the evaluator takes the top of the control stack item, executes an appropriate action, and repeats the process until the control stack is empty. The value stack contains at that moment the result of the evaluation of the expression.

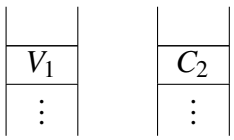
Here are the different kinds of items that can appear at the top of the control stack, and the actions taken by the evaluator.

evaluator code this is an item of type TCTRL (see § 2.3.2.15, page 13). A specific action corresponds to each of these items :

BCF00 to BCF08, BCFnn these opcodes trigger the execution of the procedure which is on the top of the value stack. **BCF00** indicates a zero-argument procedure, **BCF01** a one argument procedure, etc, up to **BCF08**. For procedures with more than 8 arguments, **BCFnn** is used. In this last case, an integer, on the top of the control stack, indicates the actual number of arguments. In the printed traces, these constants are represented as `[F0]` ,`[F1]` ,`[F2]` ,`[F3]` ,`[F4]` ,`[F5]` ,`[F6]` ,`[F7]` ,`[F8]` and `[Fn]` respectively.

It is the responsibility of the invoked procedure to pop its arguments from the value stack, and push its result onto the value stack. See for an example the § 3.1, p. 29.

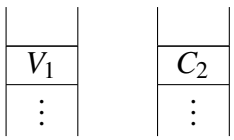
BCAND this is a conditional evaluation opcode, used in $(C_2$ `[AND]` $C_1)$, which is the compiled form of the expression `(and` E_1 E_2`)`. When this opcode is evaluated, value and control stacks have the following configuration, where C_2 is the yet unevaluated compiled form of E_2, and V_1 is the result of the evaluation of C_1 :



Value Control

The BCAND operation uses the following algorithm : if V_1 is true, this value is removed from the value stack, and execution then proceeds with C_2. If V_1 is false, the value is left on the top of stack, and C_2 is removed from the control stack.

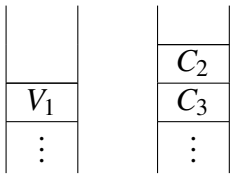
BCOR this is a conditional evaluation opcode, used in $(C_2$ `[OR]` $C_1)$, which is the compiled form of the expression `(or` E_1 E_2`)`. When this opcode is evaluated, value and control stacks have the following configuration, where C_2 is the yet unevaluated compiled form of E_2, and V_1 is the result of the evaluation of C_1 :



Value Control

The BCOR operation uses the following algorithm : if V_1 is false, this value is removed from the value stack, and execution then proceeds with C_2. If V_1 is true, the value is left on the top of stack, and C_2 is removed from the control stack.

BCIF this is a conditional evaluation opcode, used in `(`C_3 C_2 `[IF]` C_1`)`, which is the compiled form of the expression `(if` E_1 E_2 E_3`)`. When this opcode is evaluated, value and control stacks have the following configuration, where C_2 and C_3 are the yet unevaluated compiled form of E_2 and E_3, and V_1 is the result of the evaluation of C_1 :



Value Control

The BCIF operation uses the following algorithm : if V_1 is false, then C_2 is removed from the control stack ; if V_1 is true, then C_2 and C_3 are removed from the control stack, and C_2 is pushed back onto the control stack. After that, V_1 is removed from the value stack.

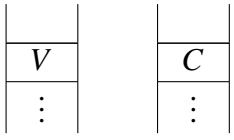
BCQUOTE this is the constant marker, used in `(`D `[QUOTE])`, which is the compiled form of the expression `(quote` D`)`. When this opcode is evaluated, the value D is on the top of the control stack. It is popped up from this stack and pushed (unchanged and unevaluated) onto the value stack.

BCPOPV this opcode suppresses the item at the top of the value stack.

BCLDENV this opcode pops a value (which must be an environment) from the control stack, and switches the evaluator to this new environment, which becomes the current environment.

BCLAMBDA this opcode creates a new function. It pops from the control stack the source code of the function, then the list of local variables, invokes the *Comp.comp* method to compile the code of the function, and pushes the result, a cell of type TLAMBDA (see § 2.3.2.12, p. 12), onto the value stack.

BCDEF this opcode creates a new binding, or modifies an existing binding. It pops a symbol or a value cell from the control stack, a value from the value stack, and defines the binding in the current environment.



Value Control

In this figure, V is the value, C the symbol or the value cell. The operation pushes the result (the symbol itself, or the *car* of the value cell, which is also a symbol) onto the value stack.

BCSET this opcode is quite similar to **BCDEF**. It pops a symbol or a value cell from the control stack, a value from the value stack, and modifies an existing binding. The result of the operation is the value itself, which is pushed onto the value stack.

BCSETU this opcode is similar to BCSET. The only difference is that it produces no result.

BCLET this opcode defines a new environment. It first determines if it is necessary to save the current environment (i.e. to push onto the control stack the current environment, then a BCLDENV opcode, a work which is not necessary when the corresponding **`let`** form is in *tail recursive position*), then creates a new empty environment, which has for parent the current environment.

BCLETREC this opcode, similar to BCLET, creates a new environment, where all the symbols defined in the corresponding **`letrec`** form are undefined, and if necessary, saves the previous environment (again, this is not necessary if the corresponding letrec form is in a *tail recursive position*).

BCFALSE this opcode replaces the item at the top of the value stack by the value *false*.

BCENDX this is the *stop code* of the evaluator. It stops immediately all interpretation, and exits with the return code *RCEND*.

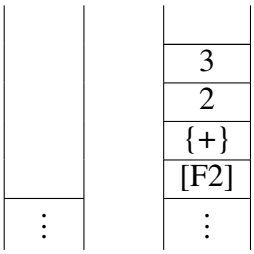
manifest constant this generic term corresponds to most of the internal data-types of MISC. Numbers, characters, booleans, nulls and strings are constants, but so are procedures, ports, environments, etc. Such an object is transfered to the top of the value stack.

value cell a value cell (see § 2.3.2.14, page 13) represents a resolved binding. The value of the cell (the *cdr* part) is transfered to the top of the value stack. It can be any Misc value (including a list), but an error is signalled (*ERRUND*) in the case of the *undefined value*. In the traces printed by the Evaluator, value cells are represented by the name of the corresponding symbol, enclosed between round brackets, as in `{var}`.

symbol a symbol represents a yet unresolved binding. The symbol is looked for in the current environment. Most of the time, the symbol is the name of a parameter of the current procedure, and is found in the immediate environment. When found in the environment, the value associated with the symbol is pushed onto the value stack. If this value is undefined, or if the symbol is not found, the *undefined value* error (ERRUND) is signalled.

list a list is a fragment of executable code, that can content any of the items described here, including other lists. The evaluator pushes the items of the list onto the *control* stack. The following pictures describe the stacks before and after such an operation, where the control stack initially holds the compiled form of the expression `(+ 2 3)`.

Value Control


3
2
{+}
[F2]
Value
Control

Chapter 3

Details of the implementation

3.1 The implementation of primitives

The primitive operations, known as *subroutines* (a reference to the very first implementation of Lisp written in FORTRAN), are implemented by classes that are subclasses of **SchPrimitive**. This class is a virtual class. Subclasses must define at least one instance, and implement the two following methods :

public static void dcl () this class method is called once, when the class is loaded, usually during the initialization of **Sch**. The purpose of this method is to declare new primitive operations of MISc.

Primitives are declared as a string array associated with one instance of the class. The string array is a field of the instance, and contains the names of the primitive procedures implemented in the class. The declaration of the new primitive is achieved by :

```
Environment.dcl(proto);
```

See for an example the class **TestOps** (§ 3.1.4, p. 31). Other simple examples are classes such as **BoolOps** or **MathsOps**.

public void eval (int op, int count, Evaluator evl) this is an instance method which is invoked each time a primitive procedure belonging to the class is called. The first parameter of the method is the number associated to the procedure. The second argument is the number of parameters of the call. The third argument is a reference to the current thread (or evaluator), which provides m`this` ethods to access parameters and return results.

3.1.1 Calling a primitive

Every primitive procedure (known as «subroutine») is represented by a cell of type *SUBR* (see § 2.3.2.7, page 11). This cell contains an object which is a representative of the class. Furthermore, the *cdr* of the cell is the local number, in the class, of the code implementing the corresponding primitive. So, calling a primitive is straightforeward ; if `opr` is a reference to such a cell, `argc` the argument count of the primitive, and `this` designating the evaluator, calling the primitive can be written :

```
int p2 = Gdm.cellcdr[opr];
Object proto = Gdm.cellobj[opr];
((SchPrimitive)proto).eval(p2,argc,this);
```

3.1.2 Classes implementing procedures

The following classes implement the primitives procedures of `Misc`. By conventions, names of the classes are of the form ***Ops**.

3.1.2.1 BoolOps

This class implements the operations on booleans, *not* and *boolean?*.

3.1.2.2 CROps

This class implements the different combinations of *car* and *cdr* defined is the Scheme standard.

3.1.2.3 ConsOps

This class implements the different procedures operating on lists and pairs : *car, cdr, cons*, etc.

3.1.2.4 EvlOps

This class implements procedures related to the *evaluators* and *lightweight processes*, *locks*, *cells* and *barriers*.

3.1.2.5 FnsOps

This class implements procedures handling other procedures : *map*, *apply* and *for-each*.

3.1.2.6 IOOps

This class implements all input/output procedures. Actual input and output operations are done by methods of classes **SchReader** and **SchPrinter**.

3.1.2.7 LangOps

This class implements all the procedures related to the internal of the language or the system : handling of environments, continuations, promises, compilation.

3.1.2.8 MathsOps

This class implements all operations on numbers.

3.1.2.9 RefOps

This class implements the reflection operations : access to java objects, classes and methods. Most of the Scheme procedures defined here have their names prefixed by «`java.`»

3.1.2.10 StringsOps

This class implements all operations on strings. Misc strings are represented by java Strings.

3.1.2.11 VectOps

This class implements the operations on vectors.

3.1.2.12 TestOps

This class can be used as a model to implement a new set of primitives in the language.

3.1.2.13 UtilsOps

This class implements some utility procedures, such as *end*, *quit*, etc.

3.1.3 Utilities

3.1.3.1 Check

This class implements, as class methods, a few utilities, like checking that an operand is an integer, a symbol, etc.

3.1.4 Implementing a primitive class

3.1.4.1 Structure of a Misc primitive

A primitive is usually implemented as a part of a `switch` in the *eval* method. Here is the class TestOps which implements a unique procedure :

```
package Sch;
import Sch.*;
 /**
   *  Demonstration of the addition of new primitives
   *  TestOps implements "test"
   *  (test) => #t
   */
public class TestOps extends SchPrimitive {
  /** This module implements 1 primitive */
  private static TestOps proto;
  public static final int OP0 = 0;
  public static final int NBOP = OP0+1;
  public static void dcl()
  {
    proto = new DemoOps();
    proto.fnames = new String[NBOP];
    proto.fnames[OP0] = "test";
    Environment.dcl(proto);
  }
  public void eval(int op, int count, Evaluator evl)
  throws SchRunTimeException
  {
    int cell, res;
```

```
        switch (op)
        {
        /** Insert new code here */
        case OP0:
            {
                if (count != 0)
                    throw new SchRunTimeException(ERRWAC);
                evl.push(CTRUE);
            }
            break;
        default :
            throw new SchRunTimeException(ERRUSP);
        }
      }
    }
```

The *eval* method here just implements one primitive, named **`test`**, which returns the value *true*. When the procedure **`test`** is used, the *eval* method is invoked by the *evaluator*, with the number associated with the procedure (here, 0), the actual number of arguments used, and a reference to the current *evaluator*. The code usually first checks that it has received a correct number of arguments. In this case, the procedure expects no arguments, and signals the error *ERRWAC* if the argument count is not zero.

Note that all errors produced during the evaluation are instances of the class **SchRunTimeException**.

3.1.4.2 Integrating a new primitive

When the new primitive class, with its methods *dcl* and *eval* is defined, it must be integrated in the MISC system, so that the new Misc primitives it defines are made available in the language. This can be done dynamically (see § 3.3, p. 34), or by including a call to the method *dcl*, either in **Sch**, in the method *schinit*, or in **Misc**, in the *main* method, after the call to *Sch.schinit* (both examples are provided in the source code).

3.2 Errors handling

This part discusses about three different aspects :

- the fact that errors may be detected at run-time by java, and that some of these errors (most, actually) must be caught so that MISC programs can be correctly processed.
- the fact that errors can be detected at run time in MISC programs, are signalled as java exceptions, and are caught at the main REPL level.
- the fact that errors in Misc programs can be caught by MISC error trapping.

3.2.1 Java run-time errors

Most Java run-time errors are caught at the level of the **Sch** main REPL. Some classes however use the specific trapping of some run-time errors, for the purpose of implementing some of the features of MISC. This part describes these specific cases.

3.2.1.1 Memory management errors

The class **Gdm** uses a specific trap to catch memory allocation error, at the end of the *gc* method. After a garbage collection has been run, the system tests if the number of cells reclaimed is large enough. If this number is less than a certain percentage of the total number of cells, it reallocates the four arrays that constitute the representation of the Misc cells. In the case of an *OutOfMemoryError,* it deallocates the new arrays and proceeds working with the old ones, and signals a memory full (*ERRNEM).*

3.2.2 Misc run-time errors

Misc run-time errors may be signaled by almost any class implementing the system. Errors are instances of the class **SchRunTimeException**, which itself is a subclass of **Exception**.

Errors may also be dynamically created by the *error* primitive procedure :

(error *string*) the procedure interrupts the current computation and prints an error message containing the string passed as a parameter to the function :

```
? (error "A message")
Error : A message
? (list 1 2 (error "an error") 3 4)
Error : an error
```

3.2.3 Trapping of Misc run-time errors

The current version of the system implements the **`try`** operation :

(try *thunk0 thunk1*) This operation executes *thunk0*, a procedure without parameter, and returns its result as the result of the *try* operation. In case of an error during the execution of *thunk0*, the error is packaged as an item, and the one parameter procedure *thunk1* is executed with the packaged error passed as parameter.

In this first example, no error arises, and the procedure *thunk1* is not called :

```
? (try (lambda () (display "Hello\n"))
?      (lambda (e) (display (string "Hum" e))))
Hello
=> "Hello\n"
```

In this second example, an error arises during the execution of *thunk0.* The procedure *thunk1* is then invoked, with the error passed as an argument.

```
? (define err)
=> err
? (try (lambda () (display "Hello\n")
?                 (+ 2 "abc") (display "Hum\n"))
?      (lambda (e) (set! err e) (display "Error found\n")))
Hello
Error found
=> "Error found\n"
? err
=> "Sch.SchRunTimeException: integer expected"
```

3.3 Dynamic loading

While new classes defining new primitives may be easily added, by integrating these classes in the MISC package, and initializing these classes from **Sch** (see § 3.1.4.2, p. 32), it is possible for a MISC program to dynamically load new primitive classes, with the *load-misc-module* operation :

(load-misc-module *string*) this operation loads the class whose name is given as a parameter of the function, and invokes the *dcl* method of the class, so that the primitives procedures of that class become available to the running MISC program. As an example, the following session illustrates the dynamic loading of a class defining the primitives `true` (which always returns `#t`) and `truth`, which returns `#f` if its parameter is #f, and `#t` otherwise :

```
[home]$ java Misc
Misc V1.1.2 Started
? true
Error : true undefined
? truth
Error : truth undefined
? (load-misc-module "Sch.TestOps")
=>  "class Sch.TestOps"
? true
=>  #<SUBR:Sch.TestOps:true>
? (true)
=>  #t
? (truth 3)
=>  #t
? (truth #f)
=>  #f
? (truth #t)
=>  #t
? (end)
; Cells use : 1225/8192, 0 gc.
Misc ended
[home]$
```

Chapter 4

Notes

4.1 Status of the work

The Misc system is available under the classical and well known GPL licence.

4.2 Work to do

In the wish list, I'll put :

1. Implementation of the quasiquote and associated constructs ;
2. A decent interface with Drew ;
3. A few other things, like having the light weight processes fully debuged, implementig the `fluid-let`, etc.
4. More coherent reification of concepts ; for example, evaluators are not reified, it is not possible to know if some class is loaded, etc.

Bibliography

[1] Jean-Jacques Girardot. Misc Internals Manual. Technical report, École des Mines, Saint-Étienne, France, 2001.

[2] Jean-Jacques Girardot. Misc User's Manual. Technical report, École des Mines, Saint-Étienne, France, 2001.

[3] R5RS. Revised ^{5} Report on the Algorithmic Language Scheme. Technical report, 1998.

[4] Christian Rolland. *LaTeX2ε, guide pratique*. Addison-Wesley, France, Juin 1995.

[5] LyX Team. Implémentation de LyX. `http://www.lyx.org/`.

Index

Contents